De l'ordonnancement des applications multi-niveaux

Vincent Lanore, Cristian Klein

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

De l'ordonnancement des applications multi-niveaux

Vincent Lanore[1], Cristian Klein[2]

[1] LIP, ENS Lyon, France
vincent.lanore@ens-lyon.fr

[2] LIP/INRIA, ENS Lyon, France
cristian.klein@inria.fr

Résumé
Sous l'impulsion des besoins applicatifs, les moyens de calcul sont de plus en plus puissants. Cette évolution se fait notamment grâce à des architectures de plus en plus parallèles. Dans ce contexte, la portabilité des performances des applications HPC très optimisées est problématique.
Le présent article est motivé par l'exemple d'une application HPC appelée HLW (High-Level Waste).
Nous présentons un modèle de la structure d'HLW indépendant de l'architecture d'exécution. Nous généralisons ensuite ce modèle en introduisant les *applications multi-niveaux* : des applications constituées d'un ensemble de tâches indépendantes ayant une probabilité de déclencher l'apparition d'une nouvelle tâche lorsqu'elles terminent. On s'intéresse ensuite à l'ordonnancement de telles tâches dans le cas où elles sont modelables, suivent la loi d'Amdahl et où l'architecture d'exécution est homogène. Nous proposons ensuite une famille d'algorithmes et évaluons ses performances à travers des simulations. Finalement, on sélectionne l'algorithme ayant les meilleures performances. Cet algorithme constitue une amélioration par rapport à l'ordonnancement par défaut d'HLW à la fois en terme de performance et d'indépendance par rapport aux paramètres de l'architecture.

Mots-clés : Ordonnancement modelable, calcul haute performance, HLW, applications multi-niveaux

1. Introduction

Afin de subvenir aux besoins croissants des applications scientifiques, les moyens de calcul augmentent d'année en année. De nouveaux types d'architectures matérielles sont mis au points et nombre d'entre eux sont parallèles. L'écriture de programmes peut être très différente selon l'architecture cible.
Les applications de calcul haute performance (en anglais *High Performance Computing* ou HPC) sont coûteuses en ressources et incluent souvent de nombreuses optimisations dont certaines dépendent de l'architecture cible. La durée de vie de ces applications étant en général bien supérieure à celle du matériel, elles doivent souvent être adaptées à de nouvelles architectures, parfois des années après leur développement originel.
Afin de résoudre ce problème, des modèles de programmation indépendants de l'architecture ont été mis au point. Ces modèles posent toutefois un nouveau problème : faire abstraction de l'architecture prive le programmeur de nombreuses possibilités d'optimisation. L'adaptation à l'architecture cible n'étant plus la responsabilité du programmeur elle revient à la plate-forme implémentant le modèle de programmation. Dans le cadre du HPC où les performances sont cruciales, ces plate-formes doivent permettre de tirer parti des possibilités d'optimisation offertes par l'architecture. Le projet COOP [1], dans le cadre duquel le travail présenté ici a été fait, s'intéresse à améliorer la capacité des modèles de programmation indépendants de l'architecture à s'adapter efficacement aux architectures cibles.
La présente étude est motivée par l'exemple d'une application HPC appelée *High-Level Waste* (HLW) qui intègre des optimisations dépendantes de l'architecture. Nous présentons tout d'abord un modèle de

1. Projet ANR 2009–2013, http ://coop.gforge.inria.fr/

performance indépendant de l'architecture pour HLW et introduisons les **applications multi-niveaux** pour généraliser la structure d'HLW. Les applications multi-niveaux sont constituées d'un ensemble de tâches indépendantes ayant une probabilité de déclencher l'apparition d'une nouvelle tâche lorsqu'elles terminent. Sur les architectures parallèles, ordonnancer ces tâches est non-trivial et est déterminant pour les performances de l'application. Le présent article s'intéresse à l'ordonnancement de telles tâches dans le cas où elles sont modelables, suivent la loi d'Amdahl et où l'architecture d'exécution est homogène (grappe de nœuds identiques reliés par un réseau homogène). On propose plusieurs algorithmes et on évalue leurs performances avec des simulations.
Cet article est organisé de la façon suivante : la section 2 présente l'application HLW, généralise sa structure en introduisant les **applications multi-niveaux** et introduit le problème d'ordonnancement associé. La section 3 présente des travaux existants en lien avec ce problème. Puis, la section 4 présente nos algorithmes qui sont évalués dans la section 5. Enfin, la section 6 conclut et ouvre des perspectives.

2. Énoncé du problème : ordonnancement des applications multi-niveaux

Cette section présente l'application HLW, propose un modèle de sa structure, introduit les applications multi-niveaux et énonce le problème de l'ordonnancement dans les applications multi-niveaux.

2.1. Présentation d'HLW

High-Level Waste (HLW) est une application HPC développée par EDF R&D pour optimiser le coût des dépôts de déchets nucléaires. C'est un problème d'optimisation sous contrainte où la fonction à optimiser est la surface du dépôt et où les contraintes sont des limites de température à certains points précis du dépôt. HLW résout ce problème par simulation numérique grâce à un programme de simulation thermique appelé *Syrthes* [2]. La logique de haut niveau d'HLW est implémentée avec Salomé [3], une plate-forme de simulation numérique développée par EDF R&D et le CEA [5].
HLW considère des dépôts dans lesquels les déchets nucléaires sont stockés dans des *conteneurs* eux-mêmes placés dans des *cellules de stockage* cylindriques et horizontales. Un tel dépôt est caractérisé par :
– τ_c, le type de conteneur ;
– D_y, la distance entre deux conteneurs consécutifs dans une même cellule de stockage ;
– P_x, la distance entre les axes de deux cellules de stockage adjacentes ;
– N_c, le nombre de conteneurs par cellule de stockage.
La fonction à minimiser est la surface de stockage E donnée comme une fonction de P_x et N_c. Pour un quadruplet de paramètres (τ_c, D_y, P_x, N_c) donné, on peut calculer la température à la surface du conteneur (en *Peau de Colis*), T_{PdC}, ainsi que la température de la terre autour des cellules (on parle de *Barrière Géologique*), T_{BG}. Un quadruplet donné de paramètres satisfait les contraintes thermiques si $T_{PdC} < T_{PdC}^{max}$ et $T_{BG} < T_{BG}^{max}$.

2.2. Modèle de l'application

HLW a une structure de type *parameter sweep* avec deux optimisations :

1. E, T_{PdC} et T_{BG} sont monotones en P_x et N_c. Ainsi, pour un (τ_c, D_y) donné il est possible d'optimiser P_x puis N_c au lieu de parcourir toutes les valeurs possibles de (P_x, N_c). En faisant deux optimisations séquentielles chacune en $O(n)$ plutôt qu'un parcours exhaustif en $O(n^2)$, la complexité d'HLW passe de $O(n^4)$ à $O(n^3)$ mais les différents calculs ne sont plus tous indépendants.
2. HLW optimise la granularité du maillage. Utiliser un maillage à grain fin donne toujours une précision acceptable. Il est toutefois possible d'utiliser un maillage à gros grain, plus rapide à calculer (environ 15 fois) et qui donne une précision acceptable la majorité du temps (deux fois sur trois). Afin de réduire le temps de calcul, chaque calcul Syrthes est d'abord fait sur un maillage à gros grain et si la précision est trop faible il est relancé sur un maillage à grain fin.

Établissons maintenant un modèle de la structure de l'application. Commençons par un modèle simple ne tenant pas compte de la seconde optimisation :
– il y a un ensemble de tâches τ_0 (les optimisations séquentielles) à effectuer ; il y a une tâche τ_0 pour chaque valeur possible de (τ_c, D_y) ;

2. http ://research.edf.com/research-and-the-scientific-community/software/syrthes-44340.html
3. http ://www.salome-platform.org/

– les tâches τ_0 sont indépendantes ;
– quand toutes les tâches τ_0 ont été effectuées, l'application termine.

Ce niveau de détail est celui auquel l'ordonnancement par défaut d'HLW opère, c'est pourquoi on le présente ici. Toutefois, nous avons décidé d'utiliser un modèle plus fin afin d'avoir plus d'opportunités d'optimisation. Dans ce second modèle, les tâches τ_0 sont plus détaillées. Soit T une tâche τ_0 :
– T consiste en $\mathrm{opt}(T)$ étapes à effectuer séquentiellement ;
– une étape commence par une tâche τ_1 qui a un temps de calcul séquentiel w_1 (calcul gros grain) ;
– une tâche τ_1 a une probabilité P_{τ_2} d'être suivie par une tâche τ_2 qui a un temps de calcul séquentiel w_2 (calcul à grain fin) ;
– une fois que la tâche τ_1 et l'éventuelle tâche τ_2 ont été effectuées, l'étape est terminée ;
– des tâches τ_1 ou τ_2 issues de tâches τ_0 différentes peuvent être effectuées indépendamment ;
– quand toutes les étapes de T ont été effectuées, T est terminée.

En dehors d'un min et d'un max, on n'a pas de données sur la distribution des $\mathrm{opt}(T)$. On considérera par défaut que $\mathrm{opt}(T)$ est une variable aléatoire entière uniforme comprise entre opt_{min} et opt_{max}.

Dans ce modèle d'HLW, les tâches ont une propriété : lorsqu'une tâche termine elle a une probabilité d'en créer une nouvelle. On appelle **multi-niveaux** une application dont les tâches sont indépendantes et ont cette propriété. À notre connaissance, ce type d'application n'a pas été étudié précédemment.

2.3. Modèle d'exécution

Le modèle que nous venons de présenter est indépendant de l'architecture : il ne capture pas l'impact de l'architecture d'exécution sur les performances de l'application.

Dans le présent article on s'intéressera au cas particulier suivant :
– Les tâches sont *modelables* ce qui signifie qu'elles peuvent se voir allouer plusieurs nœuds de calcul lorsqu'elles débutent mais ne peuvent pas voir cette allocation changer au cours de leur exécution. On notera que les tâches Syrthes sont effectivement modelables.
  On utilise la *loi d'Amdahl* pour modéliser le temps de calcul d'une tâche. Toute autre loi vérifiant l'hypothèse de monotonicité telle qu'énoncée dans [2] aurait également pu faire l'affaire. Soit $p(q)$ le temps de calcul d'une tâche T sur q noeuds de calcul et π la proportion de T qui est parallélisable (le *degré de parallélisme*). La loi d'Amdahl dit que :
$$p(q) = \left((1-\pi) + \frac{\pi}{q}\right) \times p(1)$$
– L'architecture d'exécution est composée de N nœuds de calcul identiques. On néglige les coûts de communication.

2.4. Problème d'ordonnancement

Ordonnancer les tâches d'une application multi-niveaux est un cas particulier d'*ordonnancement on-line* où c'est la fin des tâches qui provoque l'arrivée de nouvelles tâches.

Dans le présent article, on s'intéressera à minimiser le maximum des temps de complétion (temps auquel la dernière tâche termine) qui est noté C_{max}. Cette fonction objectif capture bien la performance de l'application telle que perçue par l'utilisateur.

3. Travaux connexes

Différents travaux s'intéressent à l'ordonnancement modelable on-line d'un point de vue théorique. Notre problème est noté moldable|on-line|C_{max}. Il s'agit d'un problème NP-difficile pour lequel certains algorithmes d'approximation sont toutefois connus, par exemple *l'ordonnancement par phases* [2].

Diverses heuristiques d'ordonnancement modelable on-line ont été mises au point dans le domaine de l'ordonnancement par lots. [7] et [6] discutent les méthodes existantes et proposent de nouveaux algorithmes. C_{max} n'est pas pertinent dans ce contexte et ces algorithmes optimisent d'autres fonctions objectif comme le *turnaround time*. [6] présente la borne *fair-share* qui est une borne supérieure sur le nombre de nœuds alloués à une tâche unique. Soit T une tâche et $w(T)$ son temps de calcul séquentiel. Sa borne fair-share au temps t est :
$$q_{FS}(T) = \frac{w(T)}{\sum_{T' \in A(t)} w(T')} \times N$$

où $A(t)$ est l'ensemble des tâches disponibles au temps t et N est le nombre total de nœuds de calcul.
Un problème d'ordonnancement encore plus général est celui du *parallélisme mixte*. Dans ca cas il y a à la fois des tâches modelables et des dépendances entre les tâches. L'ordonnancenement off-line de telles applications est souvent accompli avec des algorithmes en deux étapes : on alloue les nœuds aux tâches puis on fait un ordonnancement rigide classique. [4] présente CPA qui est un représentant typique de cette famille d'algorithmes, [1] et [3] présentent des variantes spécialisées de CPA. Des algorithmes en une seule étape existent et `iCASLB`, présenté dans [8], en est un bon exemple.
Tous les algorithmes présentés dans cette section pourraient être adaptés à notre problème. Toutefois, ils sont conçus pour des cas plus généraux et ne sont pas optimisés pour les applications multi-niveaux. Bien que rien n'indique a priori que ces algorithmes ne sont pas efficaces, nous pensons qu'il est possible de tirer parti des particularités des applications multi-niveaux pour concevoir des algorithmes plus performants. On se propose donc de mettre au point une famille d'algorithmes spécialisés pour les applications multi-niveaux et de vérifier qu'ils sont plus efficaces à l'aide de simulations.

4. Algorithmes proposés

Cette section présente une famille d'algorithmes pour ordonnancer les tâches dans les applications multi-niveaux. Ces algorithmes sont tous des variantes du même algorithme de base que l'on note `FS`.

4.1. L'algorithme `FS`

C'est un algorithme en deux étapes : le nombre de nœuds à allouer à chaque tâche est choisi puis un ordonnancement rigide par liste est fait. On ordonnance en priorité les tâches les plus longues car il est probable que ce soient les dernières à terminer et car les tâches courtes pourront facilement occuper l'espace laissé libre. Des expériences préliminaires confirment que cette politique tend à améliorer C_{max}.
On utilise telle quelle la borne fair-share (voir la section 3) dans `FS` pour déterminer le nombre de nœuds à allouer à chaque tâche. Le fair-share a la propriété suivante : si les tâches modelables passent parfaitement à l'échelle, si tous les nœuds sont libres au temps t et si les erreurs d'arrondi sont négligeables, alors toutes les tâches disponibles sont ordonnancées à t et finissent en même temps. Bien que cette propriété ne s'applique pas en pratique elle montre que le fair-share a tendance à homogénéiser le temps de calcul de toutes les tâches.

4.2. Améliorations

Des tests préliminaires ont permis d'identifier quelques problèmes avec l'heuristique de base que l'on vient de présenter. On proposer des améliorations afin de résoudre ces problèmes.

4.2.1. Allouer les nœuds de calcul inutilisés

À cause de l'arrondi de la formule de fair-share, certains nœuds peuvent être inutilisés après l'ordonnancement. On propose l'optimisation suivante : après avoir ordonnancé les tâches, tant qu'il reste des nœuds inutilisés, on alloue un nœud de plus à la tâche nouvellement ordonnancée avec le plus long temps de calcul. Cette optimisation est notée avec un `X` (pour "eXtra nodes"), par exemple `FSX`.

4.2.2. Retarder l'ordonnancement pour prendre de meilleures décisions

Des tests préliminaires ont montré que le fair-share a tendance à produire des ordonnancements localement mauvais lorsque le nombre de nœuds disponibles au moment de la décision d'ordonnancement est petit devant le nombre de nœuds alloués par tâche. En effet, la formule du fair-share ne tient pas compte du nombre de nœuds disponibles au moment de l'ordonnancement et fonctionne de façon optimale lorsque tous les nœuds sont disponibles.
Attendre la fin des tâches en cours presque terminées avant de prendre des décisions d'ordonnancement rendrait cette situation plus probable. De plus, plus on attend plus on a d'information au moment d'ordonnancer. On propose d'implémenter cette idée d'optimisation en introduisant le *temps de synchronisation* δ. Un ordonnanceur qui utilise cette optimisation ne peut ordonnancer une tâche au temps t que si il n'y a pas de tâche déjà ordonnancée qui termine entre t et $t + \delta$.
Il y a plusieurs façons de choisir la valeur de δ. On peut par exemple choisir $\delta = d$ où d est une constante. On note cette optimisation en ajoutant d au nom de l'algorithme, par exemple `FS0.5`.

Si δ est plus grand que le temps de calcul de la plus petite tâche disponible cela signifie que l'ordonnanceur peut laisser un espace qui aurait été suffisant pour y placer au moins une tâche. On peut résoudre ce problème en choisissant δ proportionnel à la taille de la plus petite tâche disponible. On note cette optimisation avec k`mP`, par exemple `FS0.5mP`, où k est la constante de proportionnalité.

5. Évaluation

Pour évaluer nos algorithmes et les confronter à l'ordonnancement par défaut d'HLW, un simulateur piloté par des évènements a été implémenté et une série de simulations d'ordonnancement a été faite.

5.1. Algorithmes implémentés

Afin d'estimer la pertinence des différentes améliorations proposées pour `FS`, nous avons implémenté plusieurs variantes de `FS`. Lorsque l'on choisit δ il faut s'assurer de prendre une valeur inférieure à w_1 (temps d'exécution séquentiel d'une tâche τ_1) afin que l'ordonnanceur ne laisse pas d'espace qu'une tâche τ_1 aurait pu remplir. Cette condition étant remplie, il faut une valeur aussi grande que possible afin de capturer au maximum les écarts entre les tâches qui terminent. Des tests préliminaires ont montré que les δ entre $0.3 \times w_1$ et $0.7 \times w_1$ donnaient de bons résultats. Il n'y a pas de grandes variations à l'intérieur de cet intervalle. Dans la suite on prendra $\delta = 0.5 \times w_1$. On a fait des simulations avec `FS`, `FS0.5`, `FS0.5X` en plus de l'algorithme avec toutes les optimisations `FS0.5mPX`.

Plusieurs algorithmes de référence sont comparés à nos algorithmes.

- Il n'est pas impossible qu'un **ordonnancement aléatoire** donne de bons résultats avec notre problème. On utilise comme référence l'algorithme qui choisit un nombre aléatoire de nœuds à allouer à chaque tâche et l'ordonnance au plus tôt. On note cet algorithme `rand`.
- L'**ordonnancement par défaut d'HLW** est en fait une boucle parallèle sur (τ_c, D_y) (voir section 2) distribuée sur 4 nœuds. Cela signifie que toutes les étapes d'une optimisation séquentielle donnée sont faites sur le même nœud de calcul. Cette boucle parallèle distribue les itérations de façon dynamique. Pour généraliser cet algorithme pour N nœuds, on peut décider d'ordonnancer chaque optimisation séquentielle soit sur $1/4$ des nœuds (parallélisme 4). soit sur 1 nœud (parallélisme N) On note ces algorithmes `Ref4` et `RefN` respectivement.
- Un **algorithme clairvoyant** connaît à l'avance le résultat de tous les évènements aléatoires (arrivée de tâches τ_2 et longueur des optimisations séquentielles) et peut donc faire un ordonnancement off-line. Dans ce cas, il y a des dépendances entre les tâches. Il s'agit donc d'un problème de parallélisme mixte. Un tel algorithme peut être comparé avec des algorithmes classiques afin d'estimer l'importance de prédire ces résultats. On a implémenté `CPA` (voir section 3) comme algorithme de référence.

5.2. Paramètres de l'application et des ressources

Les valeurs des paramètres pour le modèle de l'application ont été choisies proches des valeurs réelles d'HLW. On choisit comme unité de temps le temps de calcul séquentiel d'une tâche Syrthes à gros grain :

- $w_1 = 1$, le temps de calcul des tâches Syrthes à gros grain ;
- $w_2 = 15$, le temps de calcul des tâches Syrthes à grain fin ;
- $P_{\tau_2} = 0.33$, probabilité d'avoir besoin d'un calcul à grain fin ;
- $opt_{min} = 1$ et $opt_{max} = 60$, bornes pour la taille des optimisations séquentielles.

On utilise le même degré de parallélisme pour toutes les tâches (gros grain et grain fin), on le note π. On utilise la valeur $\pi = 0.99$ par défaut et on étudie l'impact de π sur les simulations (voir section 5.4).

On note N le nombre de nœuds de calcul dans l'architecture et n le nombre de tâches τ_0 à traiter.

5.3. Métriques

Afin de pouvoir comparer plus facilement les résultats on normalise les valeurs de C_{max} par rapport à une borne inférieure théorique. Une telle borne inférieure peut être obtenue en supposant qu'il n'y a aucune ressource inutilisée et que les tâches sont parfaitement parallélisables ($\pi = 1$) :

$$C_{max} \geq \frac{\sum_{T \in \text{toutes les tâches}} w(T)}{N}$$

Dans la suite, on parle de C_{max} *normalisé*.

On considère également une métrique appelée *remplissage* qui est le rapport entre la surface (temps de calcul multiplié par le nombre de nœuds alloués) de toutes les tâches et la surface totale de l'ordonnan-

cement ($C_{max} \times N$). Le remplissage quantifie l'utilisation effective des ressources pendant l'exécution.

5.4. Une sélection de simulations

Le présent article présente une sélection de simulations mise au point à partir de tests préliminaires et conçue pour mettre en valeur certains aspects clés des performances des algorithmes. Les simulations se concentrent sur deux paramètres : n/N et π.

- **Le rapport n/N** détermine le rapport au temps t entre le nombre de tâches disponibles et le nombre de nœuds disponibles. Des tests préliminaires ont montré que des expériences avec le même π et le même n/N ont tendance à produire des résultats similaires, quelle que soit la valeur de n ou N.
- **Le parallélisme** π détermine l'impact de la dilatation des tâches provoquée par la loi d'Amdahl.

5.5. Résultats et analyse

La figure 1 présente les résultats de nos simulations. Les figures 1a à 1f présentent les distributions de C_{max} normalisés pour différents scénarios. Le tableau 1g rappelle les caractéristiques des algorithmes testés. Le tableau 1h présente l'amélioration, en terme de médiane de la distribution des C_{max}, de `FS0.5mPX` par rapport au meilleur algorithme de référence pour chaque scénario. La présente section présente en détail les résultats et les analyse.

5.5.1. Une première sélection

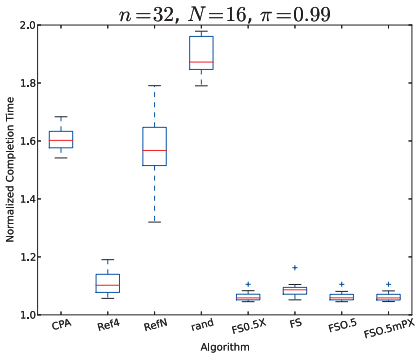
Un premier ensemble de simulations (figure 1a) a été fait avec des valeurs intermédiaires des paramètres : $n/N = 2$ et $\pi = 0.99$. Bien qu'on ne soit pas dans un scénario extrême, plusieurs des algorithmes de référence ont des performances significativement mauvaises : `CPA` et `rand`. La mauvaise performance de `CPA` peut s'expliquer par le fait que le parallélisme mixte est un problème trop général et que `CPA` a été testé pour des situations avec beaucoup plus de dépendances [4].
Les autres simulations ne gardent que `Ref4` et `RefN` comme algorithmes de référence.

5.5.2. Impact de n/N

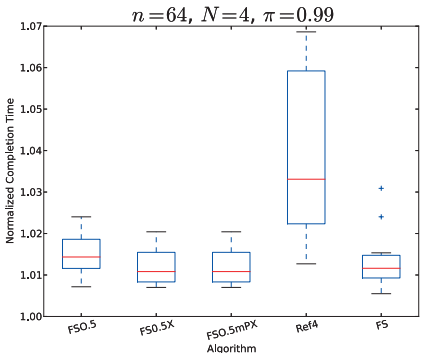
Dans la figure 1b, n/N est élevé ($n/N = 16$). Ce scénario est favorable à `FS` quelle que soit la variante. Ceci peut s'expliquer par le fait qu'au début de l'ordonnancement `FS` alloue peu de nœuds par tâche (peu de dilatation des tâches) et à l'approche de la fin va étaler les quelques tâches restantes sur les ressources disponibles. `Ref4` donne de moins bons résultats car il alloue trop de nœuds par tâche au début ce qui provoque des dilatations inutiles et que les nœuds de calcul ont tendance à être inutilisés à la fin. `Ref4` and `RefN` donnent le même ordonnancement puisque $N = 4$.
Dans la figure 1c, n/N est faible ($n/N = 1/8$). Dans ce cas, la plupart des variantes de `FS` ont des performances moins bonnes que celles de `Ref4` et seul `FS0.5mPX` est meilleur que `Ref4`. Cela peut s'expliquer par le fait que, quand n/N est faible, `FS` va avoir tendance à allouer beaucoup de nœuds par tâche et donc va diminuer le temps de calcul moyen des tâches, ce qui est un problème quand δ est une constante (voir 4.2.2). La variante $\delta = 0$ (`FS`) donne également un mauvais remplissage car les tâches vont se voir allouer beaucoup de nœuds et que l'algorithme n'attendra pas que les tâches se terminent avant d'essayer d'en ordonnancer d'autres. Ces simulations montrent que `FS0.5mPX` constitue une amélioration significative par rapport aux autres variantes. `RefN` donne de très mauvais résultats dans ce cas car la plupart des nœuds ne sont jamais utilisés.

5.5.3. Degré de parallélisme

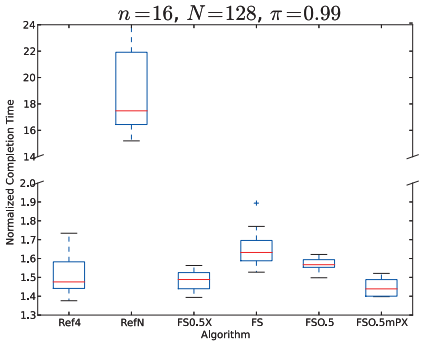
Dans la figure 1d, π et n/N sont faibles ($\pi = 0.9$ et $n/N = 1/8$). Dans ce cas, tous les algorithmes sont loin de la borne théorique. `FS` a tendance à allouer plus de nœuds par tâches quand n/N est faible. Comme π est faible, cela implique une forte dilatation des tâches et donc de mauvaises performances. Ici encore, `RefN` donne de très mauvais résultats car la plupart des nœuds ne sont jamais utilisés.
Dans la figure 1f, π est faible et n/N est élevé ($\pi = 0.9$ et $n/N = 4$). `RefN` et `Ref4` donnent tous deux de mauvais résultats comparés à `FS` et ses variantes. `Ref4` a une allocation fixe pour les tâches ce qui provoque une importante dilatation à cause du faible parallélisme là où `FS` et ses variantes allouent peu de nœuds par tâche puisque n/N est élevé. `RefN` n'a pas de problème de dilatation des tâches mais utilise mal les ressources en fin d'ordonnancement.
Dans la figure 1e, le parallélisme est parfait et n/N est faible ($\pi = 1$ et $n/N = 1/8$). Les variantes de `FS` autres que `FS0.5mPX` donnent des résultats moins bons que `Ref4`. Ce scénario est globalement similaire



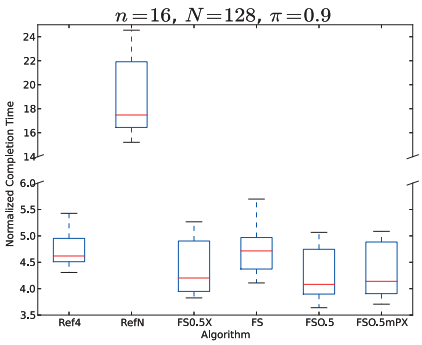
(a) Distribution du C_{max} normalisé pour des valeurs moyennes des paramètres.



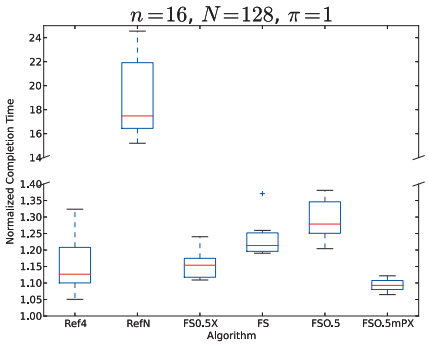
(b) Distribution du C_{max} normalisé pour un n/N élevé et un π moyen.



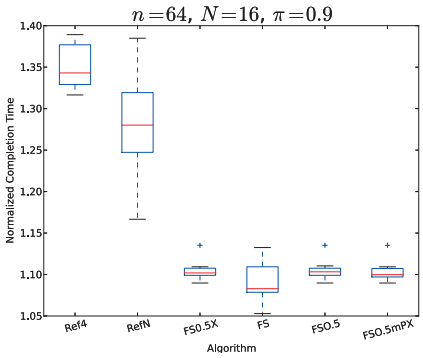
(c) Distribution du C_{max} normalisé pour un n/N faible et un π moyen.



(d) Distribution du C_{max} normalisé pour un n/N faible et un π faible.



(e) Distribution du C_{max} normalisé pour un n/N faible et un π élevé.



(f) Distribution du C_{max} normalisé pour un n/N élevé et un π faible.

Algorithmes FS			Algorithmes de référence	
Nom	δ	X	Nom	Caractéristiques
FS	0	non	rand	Allocation nombre aléatoire de noeuds
FS0.5	$0.5 \times w_1$	non	CPA	Algorithme clairvoyant ; parallélisme mixte
FS0.5X	$0.5 \times w_1$	oui	Ref4	Ordonnancement HLW, parallélisme 4
FS0.5mPX	$k = 0.5$	oui	RefN	Ordonnancement HLW, parallélisme N

(g) Caractéristiques des algorithmes testés.

		n/N		
		low	med	high
π	low	11%	-	22%
	med	3%	4%	2%
	high	-	-	4%

(h) Amélioration apportée par FS0.5mpX par rapport au meilleur algorithme de référence pour chaque scénario, en terme de médiane de la distribution de C_{max}.

FIGURE 1 – Performances des algorithmes de la famille FS comparés à des algorithmes de référence.

au cas où n/N est faible vu plus haut. On notera tout de même que les variantes avec un δ fixe ont des performances moins bonnes encore puisque le temps de calcul des tâches est plus faible. À nouveau, `FS0.5mPX` donne des résultats significativement meilleurs que ceux de `Ref4`.

5.6. Discussion

Les simulations montrent que, quels que soient les paramètres de simulation, `FS0.5mPX` est le meilleur algorithme pour notre problème (Table 1g). Selon le scénario, l'amélioration en terme de performance varie entre 2% et 22%. De plus, `FS0.5mPX` donne de bons résultats quels que soient le scénario là où les algorithmes de référence ont chacun au moins un scénario dans lequel ils donnent de mauvais résultats.

6. Conclusion

Le présent article s'est intéressé à une application HPC appelée HLW, a proposé un modèle pour sa structure et a introduit les *applications multi-niveaux*. On s'est ensuite intéressé à l'ordonnancement des tâches dans de telles applications dans les cas où ces tâches sont modelables, suivent la loi d'Amdahl et où l'architecture est homogène. On a proposé des heuristiques pour résoudre ce problème d'ordonnancement que l'on a validées avec des simulations en les comparant à divers algorithmes de référence. Nos algorithmes donnent de bonnes performances sur les architectures homogènes, quelle que soit leur taille. Cela constitue une amélioration par rapport à l'ordonnancement par défaut d'HLW à la fois en matière de performance et d'indépendance vis-à-vis des paramètres.

Nous ne pensons pas qu'il soit possible de faire une analyse théorique de notre algorithme notamment en raison du caractère irrégulier des optimisations proposées.

Un première extension possible de ce travail serait de faire des expérience avec la véritable application HLW afin de confirmer l'utilité de nos algorithmes dans un cas réel d'application HPC. Il y a des travaux en cours visant à faire fonctionner HLW sur la plate-forme Grid'5000.

Une autre extension possible serait de voir comment nos algorithmes pourraient être intégrés dans des modèles de programmation pour le calcul haute performance. HLW est écrit avec la plate-forme Salomé et utilise un modèle de programmation appelé YACS. Dans son état actuel, YACS ne permet pas d'intégrer facilement des algorithmes comme celui que nous proposons.

Enfin, notre algorithme pourrait être adapté à des modèles de ressources plus généraux. Dans un contexte de grille ou de cloud par exemple, les ressources disponibles peuvent changer au cours de l'exécution. Notre simulateur et nos algorithmes pourraient aisément être réutilisés pour étudier de tels problèmes.

Bibliographie

1. Desprez (F.) et Suter (F.). – A bi-criteria algorithm for scheduling parallel task graphs on clusters. *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010)*, 2010.
2. Dutot (P.-F.), Mounie (G.) et Trystram (D.). – Scheduling parallel tasks — approximation algorithms. *In : Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. – CRC Press, 2004.
3. N'Takpe (T.), Suter (F.) et Casanova (H.). – A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms. *Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, 2007.
4. Radulescu (A.) et van Gemund (A. J.). – A low-cost approach towards mixed task and data parallel scheduling. *Proc. of 2001 International Conference on Parallel Processing*, 2001.
5. Ribes (A.) et Caremoli (C.). – Salomé platform component model for numerical simulation. *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, 2007.
6. Sabin (G.), Lang (M.) et Sadayappan (P.). – Moldable parallel job scheduling using job efficiency : An iterative approach. *Workshop on Job Scheduling Strategies for Parallel Processing*, 2006.
7. Saule (E.), Bozdağ (D.) et Catalyurek (U. V.). – A moldable online scheduling algorithm and its application to parallel short sequence mapping. *JSSPP'10 Proceedings of the 15th international conference on Job scheduling strategies for parallel processing*, 2010.
8. Vydyanathan (N.), Krishnamoorthy (S.), Sabin (G.), Catalyurek (U.), Kurc (T.), Sadayappan (P.) et Saltz (J.). – An integrated approach for processor allocation and scheduling of mixed-parallel applications. *Parallel Processing, 2006. ICPP 2006*, 2006.